CAPITOLO 11

Il package di input e output

di Michael Morrison

IN QUESTO CAPITOLO

Un programma non potrebbe fare nulla di utile senza effettuare alcun tipo di input o di output dei dati. La maggior parte dei programmi necessita dell'input dell'utente e, in cambio, restituisce informazioni sullo schermo, tramite la stampante e spesso su file. Il package di I/O di Java fornisce un'ampia gamma di classi che gestiscono l'input e l'output da e in diversi dispositivi. In questo capitolo vengono descritte le classi principali contenute nel package di I/O, assieme ad alcuni esempi che mostrano le funzionalità di queste classi.

Il package di I/O, `java.io`, contiene molte classi, ognuna delle quali ha diversi metodi e variabili membro. Questo capitolo non contiene una descrizione dettagliata di tutte le classi e di tutti i metodi inclusi nel package di I/O; si può dire invece che rappresenta una guida su come eseguire le operazioni fondamentali di input e output di base utilizzando le classi più comuni, in modo da poter iniziare a utilizzare le classi di I/O di Java nei programmi. Questo capitolo inoltre pone le basi per chi decidesse di esplorare le classi di I/O più complesse supportate in Java.

Classi del flusso di input e classi di lettura

Il modello di input di Java si basa sul concetto di *flusso di input*, che può essere considerato come un flusso fisico di acqua che scorre nei tubi di un sistema idrico. La differenza ovvia è che un flusso di input gestisce dati binari del computer, e non l'acqua fisica. Il confronto tuttavia è valido, in quanto i dati in un flusso di input scorrono come l'acqua che viene pompata in un tubo. I dati spinti in un flusso di input possono essere indirizzati in diversi

modi, come l'acqua che viene indirizzata in un complesso impianto di tubi che compongono un sistema idrico. I dati in un flusso di input vengono trasmessi un byte alla volta, più o meno come singole gocce di acqua che scorrono in un tubo.

In termini più pratici, Java utilizza i flussi di input come strumento di lettura dei dati da una *sorgente di input*, ad esempio la tastiera. Le classi principali per il flusso di input supportate in Java sono:

- `InputStream`
- `BufferedInputStream`
- `DataInputStream`
- `FileInputStream`
- `StringBufferInputStream`

Con la versione 1.1 di Java è stato introdotto il supporto per i *flussi di input di caratteri*, che sono virtualmente identici ai flussi di input, a eccezione del fatto che agiscono sui caratteri anziché sui byte. Le classi del flusso di input di caratteri sono dette *di lettura*, anziché di input. Le classi di lettura implementano metodi simili a quelli delle classi del flusso di input corrispondenti elencate in precedenza, a eccezione della classe `DataInputStream`. Lo scopo delle versioni che si basano sui caratteri delle classi del flusso di input è quello di facilitare l'internazionalizzazione. Le classi di lettura principali incluse in Java sono:

- `Reader`
- `BufferedReader`
- `FileReader`
- `StringReader`

In questo capitolo vengono utilizzate sia le classi del flusso di input sia le classi di lettura. Poiché i metodi supportati da queste classi sono molto simili, risulta semplice utilizzare le une dopo aver appreso le altre.

La classe InputStream

`InputStream` è una classe astratta che serve come classe di base per tutte le altre classi del flusso di input; definisce un'interfaccia di base per la lettura dei flussi di byte di informazioni. I metodi definiti in questa classe diventeranno molto familiari, in quanto vengono utilizzati per lo stesso scopo in tutte le classi derivate da `InputStream`. Questo approccio di progettazione permette di apprendere il protocollo per la gestione dei flussi di input una volta sola e di applicarlo a diversi dispositivi utilizzando una classe derivata da `InputStream`.

Lo scenario tipico, quando si utilizza un flusso di input, consiste nel creare un oggetto derivato da `InputStream` e quindi informarlo che si desidera ricevere informazioni, richiamando un metodo appropriato. Se non sono disponibili informazioni di input, `InputStream`

utilizza una tecnica detta *bloccaggio* per rimanere in attesa finché diventano disponibili dei dati di input. Un esempio di bloccaggio si ha quando si utilizza un flusso di input per leggere le informazioni dalla tastiera: finché l'utente digita le informazioni e non preme il tasto Invio, non vi è input disponibile per l'oggetto `InputStream`, che rimane quindi in attesa finché l'utente preme il tasto Invio; a questo punto i dati diventano disponibili e l'oggetto `InputStream` può elaborarli come input.

La classe `InputStream` definisce i seguenti metodi:

- `abstract int read()`
- `int read(byte` *b*`[])`
- `int read(byte` *b*`[], int` *off*`, int` *lun*`)`
- `long skip(long` *n*`)`
- `int available()`
- `synchronized void mark(int` *limiteLettura*`)`
- `synchronized void reset()`
- `boolean markSupported()`
- `void close()`

`InputStream` definisce tre diversi metodi `read()` per leggere i dati di input in diversi modi. Il primo metodo `read()` non utilizza parametri, legge semplicemente un byte di dati dal flusso di input e lo restituisce come intero. Questa versione di `read()` restituisce `-1` se viene raggiunta la fine del flusso di input; poiché un byte di input viene restituito come `int`, se si leggono caratteri è necessario eseguire il casting in un `char`. La seconda versione di `read()` utilizza come unico parametro un array di byte, cosa che permette di leggere diversi byte di dati alla volta. I dati letti vengono memorizzati in questo array. È necessario assicurarsi che l'array di byte passato in `read()` sia sufficientemente grande da contenere le informazioni che vengono lette, altrimenti viene generata un'eccezione `IOException`. Questa versione di `read()` restituisce il numero effettivo di byte letti o `-1` se viene raggiunta la fine del flusso. L'ultima versione di `read()` utilizza come parametri un array di byte, un offset di interi e una lunghezza intera. Questa versione di `read()` è simile alla seconda, a eccezione del fatto che permette di specificare in quale punto dell'array di byte si desidera inserire le informazioni lette. Il parametro *off* specifica l'offset nell'array di byte in cui iniziare a inserire i dati letti, mentre il parametro *lun* specifica il numero massimo di byte da leggere.

Il metodo `skip()` viene utilizzato per saltare dei byte di dati nel flusso di input. `skip()` utilizza come unico parametro un valore `long` *n*, che specifica quanti byte di input saltare e restituisce il numero effettivo di byte saltati o `-1` se viene raggiunta la fine del flusso di input.

Il metodo `available()` viene utilizzato per determinare il numero di byte dei dati di input che possono essere letti senza che venga posto in atto il bloccaggio. `available()` non utilizza parametri e restituisce il numero di byte disponibili. Questo metodo è utile se si desidera

garantire che vi siano dei dati di input disponibili (evitando di conseguenza il meccanismo di bloccaggio).

Il metodo `mark()` segna la posizione corrente nel flusso. È possibile tornare successivamente a questa posizione utilizzando il metodo `reset()`. I metodi `mark()` e `reset()` sono utili in situazioni in cui si desidera leggere i dati successivi nel flusso senza perdere la posizione originale, ad esempio quando si verifica il tipo di un file di immagine. In questo caso è possibile leggere prima il titolo del file e segnare la posizione alla fine del titolo stesso, quindi leggere dei dati per assicurarsi che rispettino il formato previsto per quel tipo di file. Se i dati non sembrano corretti, si può tornare al punto segnato e provare una tecnica diversa.

Si noti che il metodo `mark()` utilizza come parametro un intero, *`limiteLettura`*, che specifica quanti byte possono essere letti prima che il segno diventi invalidato. In realtà *`limiteLettura`* determina fin dove si può arrivare con la lettura rimanendo in grado di tornare alla posizione segnata. Il metodo `markSupported()` restituisce un valore booleano che indica se il flusso di input supporta le funzionalità di `mark()` e di `reset()`.

Infine, il metodo `close()` chiude un flusso di input e rilascia tutte le risorse associate. Non è necessario richiamare esplicitamente `close()`, in quanto i flussi di input vengono chiusi automaticamente quando viene distrutto l'oggetto `InputStream`. Anche se non è necessaria, la chiamata a `close()` subito dopo aver terminato di utilizzare un flusso è una buona pratica di programmazione, in quanto `close()` fa in modo che un il buffer del flusso sia svuotato, cosa che contribuisce a evitare il danneggiamento del file.

L'oggetto System.in

La tastiera è il dispositivo standard per l'input dell'utente. La classe `System` inclusa nel package del linguaggio contiene una variabile membro che rappresenta la tastiera (il flusso di input standard). Questa variabile membro è chiamata `in` ed è un'istanza della classe `InputStream`, utile per leggere l'input dell'utente tramite la tastiera. Il Listato 11.1 contiene il programma `ReadKeys1`, che mostra come sia possibile connettere l'oggetto `System.in` a un oggetto `InputStreamReader` per leggere l'input dell'utente tramite la tastiera. Questo programma si trova nel file `ReadKeys1.java` nel CD-ROM allegato al libro.

La tastiera è stata menzionata come flusso di input standard. Ciò non è completamente vero, in quanto il flusso di input standard può ricevere l'input da numerose sorgenti. Nonostante la tastiera sia certamente il metodo più comune per inserire l'input nel flusso di input standard, certamente non è l'unico. Un esempio di flusso di input standard da una sorgente di input diversa è il reindirizzamento di un file di input in un flusso.

Listato 11.1 *La classe ReadKeys1.*

```
import java.io.*;
class ReadKeys1 {
  public static void main (String args[]) {
```

```
      StringBuffer s = new StringBuffer();
      char c;
      try {
        Reader in = new InputStreamReader(System.in);
        while ((c = (char)in.read()) != '\n') {
          s.append(c);
        }
      }
      catch (Exception e) {
        System.out.println("Errore: " + e.toString());
      }
      System.out.println(s);
    }
  }
```

La classe ReadKeys1 crea innanzitutto un oggetto StringBuffer chiamato s, quindi crea un oggetto di lettura connesso al flusso di input standard e inserisce un ciclo while che richiama ripetutamente il metodo read() finché non viene rilevato un carattere di nuova riga (\n), vale a dire finché l'utente non preme il tasto Invio. Si noti che i dati di input restituiti da read() vengono convertiti nel tipo char prima di essere memorizzati nella variabile carattere c. Ogni volta che viene letto un carattere, questo viene aggiunto nel buffer della stringa utilizzando il metodo append() di StringBuffer. È importante osservare come qualsiasi errore causato dal metodo read() venga gestito dai blocchi di gestione delle eccezioni try/catch. Il blocco catch stampa semplicemente un messaggio di errore nel flusso di output standard; infine, quando dal flusso di input viene letto un carattere di nuova riga, viene chiamato il metodo println() del flusso di output standard per stampare la stringa sullo schermo. Il flusso di output standard viene discusso più avanti in questo capitolo.

Il Listato 11.2 include ReadKeys2, simile a ReadKeys1, a eccezione del fatto che utilizza la seconda versione del metodo read(). Questo metodo read() utilizza come parametro un array di caratteri per memorizzare l'input letto. ReadKeys2 si trova nel file ReadKeys2.java nel CD-ROM allegato al libro.

Listato 11.2 *La classe ReadKeys2.*

```
import java.io.*;
class ReadKeys2 {
  public static void main (String args[]) {
    char buf[] = new char[80];
    try {
      Reader in = new InputStreamReader(System.in);
      in.read(buf, 0, 80);
    }
    catch (Exception e) {
      System.out.println("Errore: " + e.toString());
    }
    String s = new String(buf);
    System.out.println(s);
  }
}
```

In `ReadKeys2` viene creato un array con 80 caratteri. Viene inoltre creato e connesso al flusso di input standard un oggetto di lettura e viene eseguita un'unica chiamata al metodo `read()` che legge tutto ciò che l'utente ha digitato. L'input viene bloccato finché l'utente preme il tasto [Invio], l'input diventa disponibile e il metodo `read()` riempie l'array di caratteri con i nuovi dati. Successivamente viene creato un oggetto `String` per contenere la stringa costante creata in precedenza. Si noti che il costruttore utilizzato per creare l'oggetto `String` utilizza come primo parametro un array di caratteri (`buf`). Infine viene utilizzato nuovamente `println()` per stampare la stringa.

Il programma `ReadKeys3` nel Listato 11.3 mostra l'utilizzo dell'ultima versione del metodo `read()`, che impiega un array di caratteri e anche un offset e una lunghezza per determinare il modo in cui i dati di input vengono memorizzati nell'array di caratteri. `ReadKeys3` si trova nel file `ReadKeys3.java` nel CD-ROM allegato al libro.

Listato 11.3 *La classe ReadKeys3.*

```
import java.io.*;
class ReadKeys3 {
  public static void main (String args[]) {
    char buf[] = new char[10];
    try {
      Reader in = new InputStreamReader(System.in);
      in.read(buf, 0, 10);
    }
    catch (Exception e) {
      System.out.println("Errore: " + e.toString());
    }
    String s = new String(buf);
    System.out.println(s);
  }
}
```

`ReadKeys3` è molto simile a `ReadKeys2`, con un'unica differenza principale: la terza versione del metodo `read()` viene utilizzata per limitare il numero massimo di caratteri letti nell'array. Inoltre, le dimensioni dell'array vengono ridotte a 10 caratteri per mostrare che cosa fa questa versione di `read()` quando sono disponibili più dati di quelli che può contenere l'array. Si ricordi che questa versione di `read()` può essere utilizzata anche per leggere dati in un offset specifico dell'array. In questo caso, l'offset specificato è `0` e pertanto l'unica differenza è il numero massimo di caratteri che possono essere letti (`10`). Questa tecnica garantisce che l'array non venga sovraccaricato.

La classe BufferedInputStream

Come indica il nome, la classe `BufferedInputStream` fornisce un flusso di input bufferizzato, vale a dire che nel flusso vengono letti più dati di quelli richiesti e pertanto le letture successive vengono effettuate dal buffer anziché dal dispositivo di input. Questa tecnica permette un accesso per la lettura molto più veloce, in quanto la lettura da un buffer è semplicemente una lettura dalla memoria. `BufferedInputStream` implementa gli stessi metodi definiti da

InputStream. In realtà, non implementa nessun metodo nuovo, ma ha due diversi costruttori, indicati di seguito:

- ✔ `BufferedInputStream(InputStream` *`in`*`)`
- ✔ `BufferedInputStream(InputStream` *`in`*`, int` *`dim`*`)`

Si noti che entrambi i costruttori utilizzano come primo parametro un oggetto `InputStream`. L'unica differenza tra i due è data dalle dimensioni del buffer interno: nel primo costruttore viene utilizzato un buffer di dimensioni predefinite, mentre nel secondo le dimensioni del buffer vengono specificate con il parametro intero *dim*. Per supportare l'input bufferizzato, la classe `BufferedInputStream` definisce inoltre alcune variabili membro, indicate di seguito:

- ✔ `byte` *`buf`*`[]`
- ✔ `int` *`conta`*
- ✔ `int` *`pos`*
- ✔ `int` *`posSegno`*
- ✔ `int` *`limiteSegno`*

L'array di byte *buf* è il buffer in cui vengono memorizzati i dati dell'input. La variabile membro *conta* tiene il conto di quanti byte sono memorizzati nel buffer. La variabile membro *pos* tiene il conto della posizione di lettura corrente nel buffer. La variabile membro *posSegno* specifica la posizione correntemente segnata nel buffer impostata con il metodo `mark()` ed è uguale a `-1` se non è stato impostato alcun segno. Infine, la variabile membro *limiteSegno* specifica il numero massimo di byte che possono essere letti prima che la posizione segnata non sia più valida. *limiteSegno* viene impostata dal parametro *limiteLettura* passato nel metodo `mark()`. Poiché tutte queste variabili membro sono specificate come `protected`, probabilmente non se ne utilizzerà mai nessuna, tuttavia danno un'idea di come la classe `BufferedInputStream` implementa i metodi definiti da `InputStream`.

La classe BufferedReader è molto simile alla classe BufferedInputStream, a eccezione del fatto che gestisce caratteri anziché byte. Il Listato 11.4 include il programma ReadKeys4, che utilizza un oggetto BufferedReader al posto di un oggetto InputStreamReader per leggere l'input dalla tastiera. ReadKeys4 si trova nel file ReadKeys4.java nel CD-ROM allegato al libro.

Listato 11.4 *La classe ReadKeys4.*

```
import java.io.*;
class ReadKeys4 {
  public static void main (String args[]) {
    Reader in = new BufferedReader(new InputStreamReader(System.in));
    char buf[] = new char[10];
    try {
      in.read(buf, 0, 10);
    }
    catch (Exception e) {
```

```
        System.out.println("Errore: " + e.toString());
      }
      String s = new String(buf);
      System.out.println(s);
    }
  }
```

L'oggetto BufferedReader viene creato passando il flusso di input System.in in un oggetto *InputStreamReader*. Questo approccio è necessario, in quanto *il* costruttore BufferedReader() necessita di un oggetto derivato da Reader. Da questo punto in poi, il programma è sostanzialmente uguale a ReadKeys3, a eccezione del fatto che il metodo read() viene richiamato sull'oggetto BufferedReader anziché sull'oggetto InputStreamReader.

La classe DataInputStream

La classe DataInputStream è utile per leggere i tipi di dati primitivi di Java da un flusso di input in un modo che consente la portabilità. Esiste un solo costruttore per DataInputStream, che utilizza semplicemente come unico parametro un oggetto InputStream. Questo costruttore è definito nel seguente modo:

```
DataInputStream(InputStream in)
```

DataInputStream implementa i seguenti metodi, oltre a quelli definiti da InputStream:

- final int skipBytes(int *n*)
- final void readFully(byte *b*[])
- final void readFully(byte *b*[], int *off*, int *lun*)
- final boolean readBoolean()
- final byte readByte()
- final int readUnsignedByte()
- final short readShort()
- final int readUnsignedShort()
- final char readChar()
- final int readInt()
- final long readLong()
- final float readFloat()
- final double readDouble()

Il metodo skipBytes() funziona in modo molto simile a skip(), a eccezione del fatto che skipBytes() mantiene il blocco finché sono stati saltati tutti i byte. Il numero di byte da saltare è determinato dal parametro intero *n*. Vi sono due metodi readFully() implementati

da DataInputStream simili ai metodi read(), ma questi mantengono il blocco finché sono stati letti tutti i dati, mentre i metodi read() normali mantengono il blocco solo finché sono disponibili alcuni dati, non tutti. I metodi readFully() sono per i metodi read() quello che skipBytes() è per skip(). Gli altri metodi implementati da DataInputStream sono varianti del metodo read() per i diversi tipi di dati primitivi. Il tipo letto da ogni metodo può essere facilmente identificato dal nome del metodo stesso.

La classe FileInputStream

La classe FileInputStream è utile per eseguire operazioni di input su file semplici. Per operazioni di input su file più avanzate, si utilizza la classe RandomAccessFile, discussa più avanti in questo capitolo. La classe FileInputStream può essere istanziata utilizzando uno dei seguenti costruttori:

- ✔ FileInputStream(String *nome*)
- ✔ FileInputStream(File *file*)
- ✔ FileInputStream(FileDescriptor *fdObj*)

Il primo costruttore utilizza come parametro un oggetto String chiamato *nome*, che specifica il nome del file da utilizzare per l'input. Il secondo costruttore utilizza un parametro oggetto File che specifica il file da utilizzare per l'input (l'oggetto File viene discusso verso la fine di questo capitolo). Il terzo costruttore di FileInputStream utilizza come unico parametro un oggetto FileDescriptor.

La classe FileInputStream funziona esattamente come la classe InputStream, a eccezione del fatto che opera con i file. In modo simile, la classe FileReader funziona in modo simile alla classe FileInputStream, a eccezione del fatto che opera su caratteri anziché su byte. Il Listato 11.5 include il programma ReadFile, che utilizza la classe FileReader per leggere i dati da un file di testo. ReadFile si trova nel file ReadFile.java nel CD-ROM allegato al libro.

Listato 11.5 *La classe ReadFile.*

```
import java.io.*;
class ReadFile {
  public static void main (String args[]) {
    char buf[] = new char[64];
    try {
      Reader in = new FileReader("Grocery.txt");
      in.read(buf, 0, 64);
    }
    catch (Exception e) {
      System.out.println("Errore: " + e.toString());
    }
    String s = new String(buf);
    System.out.println(s);
  }
}
```

In ReadFile, viene prima creato un oggetto FileReader passando una stringa con il nome del file ("Grocery.txt") come file di input, quindi viene richiamato il metodo read() per leggere dal file di input in un array di caratteri. L'array di caratteri viene infine utilizzato per creare un oggetto String, che a sua volta viene utilizzato per l'output.

La classe StringBufferInputStream

A parte il nome molto lungo, StringBufferInputStream è una classe piuttosto chiara, che permette di utilizzare una stringa come sorgente bufferizzata di input. La classe StringBufferInputStream implementa gli stessi metodi definiti da InputStream, e nessun altro, e ha un unico costruttore:

```
StringBufferInputStream(String s)
```

Il costruttore utilizza un oggetto String, da cui crea il flusso di input bufferizzato. Nonostante StringBufferInputStream non definisca altri metodi, include alcune variabili membro proprie:

- String buffer
- int count
- int pos

La stringa membro buffer è il buffer in cui vengono memorizzati i dati di una stringa; la variabile membro conta specifica il numero di caratteri da utilizzare nel buffer, mentre la variabile membro pos tiene il conto della posizione corrente nel buffer. Come per la classe BufferedInputStream, probabilmente non si vedranno mai queste variabili membro, ma sono importanti per capire come è implementata la classe StringBufferInputStream.

La classe StringReader è una versione che si basa su caratteri della classe StringBufferInputStream. Il Listato 11.6 include il programma ReadString, che utilizza un oggetto StringReader per leggere i dati da una stringa di dati di testo. ReadString si trova nel file ReadString.java nel CD-ROM allegato al libro.

Listato 11.6 *La classe ReadString.*

```
import java.io.*;
class ReadString {
  public static void main (String args[]) {
    // Ottiene una stringa di input dall'utente
    char buf1[] = new char[64];
    try {
      Reader in = new InputStreamReader(System.in);
      in.read(buf1, 0, 64);
    }
    catch (Exception e) {
      System.out.println("Errore: " + e.toString());
    }
    String s1 = new String(buf1);
    // Legge la stringa e la invia in output
```

```
        Reader in = new StringReader(s1);
        char buf2[] = new char[64];
        try {
          in.read(buf2, 0, 64);
        }
        catch (Exception e) {
          System.out.println("Errore: " + e.toString());
        }
        String s2 = new String(buf2);
        System.out.println(s2);
      }
    }
```

Il programma `ReadString` permette all'utente di digitare del testo che viene letto e memorizzato in una stringa, la quale viene quindi utilizzata per creare un oggetto `StringReader` che viene letto in un'altra stringa per l'output. Ovviamente, questo programma è molto complicato ed esegue solo alcune semplici operazioni, ma è stato scritto solo come dimostrazione dell'utilizzo della classe `StringReader`.

La prima metà del programma `ReadString` dovrebbe risultare familiare, infatti si tratta in pratica del nucleo del programma `ReadKeys3`, discusso precedentemente in questo capitolo, che legge i dati immessi in una stringa tramite la tastiera. La seconda metà del programma è la parte in cui interviene l'oggetto `StringReader`. In questa parte viene infatti creato un oggetto `StringReader` utilizzando l'oggetto `String` (`s1`) che contiene il testo immesso tramite la tastiera. Il contenuto dell'oggetto `StringReader` viene quindi letto in un array di caratteri utilizzando il metodo `read()`. L'array di caratteri a sua volta viene utilizzato per creare un altro oggetto `String` (`s2`), che viene stampato sullo schermo.

Classi del flusso di output e classi di scrittura

In Java, i flussi di output sono le controparti logiche dei flussi di input e gestiscono la scrittura di dati in sorgenti di output. Utilizzando l'analogia con l'acqua già presentata nella discussione sui flussi di input, un *flusso di output* è l'equivalente del rubinetto di un lavandino. Esattamente come l'acqua, che scorre da un impianto attraverso i tubi ed esce dal rubinetto, i dati scorrono da un dispositivo di input attraverso il sistema operativo ed escono da un dispositivo di output. Un rubinetto che perde rappresenta addirittura un modo migliore per visualizzare il trasferimento di dati da un flusso di output: ogni goccia di acqua che cade dal rubinetto rappresenta un byte di dati. Ogni byte di dati scorre al dispositivo di output, esattamente come le gocce d'acqua cadono una dopo l'altra dal rubinetto.

Tornando a Java, i flussi di output vengono utilizzati per inviare dati in diversi dispositivi di output, ad esempio lo schermo. Le classi principali del flusso di output utilizzate per la programmazione in Java sono:

- ✔ `OutputStream`

- `PrintStream`
- `BufferedOutputStream`
- `DataOutputStream`
- `FileOutputStream`

I flussi di output di Java permettono di eseguire l'output dei dati in diversi modi. La classe `OutputStream` definisce il comportamento essenziale richiesto da un flusso di output, la classe `PrintStream` viene utilizzata per l'output di dati di testo, ad esempio i dati inviati al flusso di output standard, la classe `BufferedOutputStream` è un'estensione della classe `OutputStream` che fornisce il supporto per l'output bufferizzato, la classe `DataOutputStream` è utile per l'output di tipi di dati primitivi, ad esempio gli `int` o i `float`; infine, la classe `FileOutputStream` fornisce il supporto necessario per l'output di dati su file.

Java supporta anche i flussi di output di caratteri, che sono virtualmente identici ai flussi di output sopra elencati, a eccezione del fatto che agiscono su caratteri anziché su byte. I flussi di output di caratteri sono chiamati *di scrittura*, anziché di output. Le classi di scrittura implementano metodi simili a quelli delle classi del flusso di output corrispondenti, a eccezione della classe `DataOutputStream`. Lo scopo delle versioni che si basano sui caratteri è quello di facilitare l'internazionalizzazione. Ecco le classi di scrittura fondamentali di Java:

- `Writer`
- `PrintWriter`
- `BufferedWriter`
- `FileWriter`

In questo capitolo si utilizzano sia le classi del flusso di output sia le classi di scrittura. Poiché i metodi supportati da queste classi sono molto simili, risulta semplice utilizzare le une dopo aver appreso le altre.

La classe OutputStream

La classe `OutputStream` è la controparte per l'output di `InputStream` e serve come classe di base astratta per tutte le altre classi del flusso di output. `OutputStream` definisce il protocollo di base per scrivere flussi di dati in un dispositivo di output. Come nel caso dei metodi di `InputStream`, è facile acquisire familiarità con i metodi definiti da `OutputStream`, in quanto essi agiscono nello stesso modo in ogni classe derivata. Il vantaggio di questa interfaccia comune è che, dopo aver appreso un metodo una volta, si è in grado di applicarlo a diverse classi senza dover iniziare il processo da capo.

Di norma si crea un oggetto derivato da `OutputStream` e si richiama un metodo appropriato per indicargli che si desidera l'output di informazioni. La classe `OutputStream` utilizza una tecnica simile a quella di `InputStream`: mantiene il blocco finché nel dispositivo di output sono stati scritti dei dati. Durante il blocco (in attesa che l'output corrente venga elaborato), la classe `OutputStream` non permette che vengano inviati altri dati.

La classe OutputStream implementa i seguenti metodi:

- abstract void write(int *b*)
- void write(byte *b*[])
- void write(byte *b*[], int *off*, int *lun*)
- void flush()
- void close()

OutputStream definisce tre diversi metodi write() per scrivere i dati in alcuni modi diversi. Il primo metodo write() scrive un singolo byte alla volta nel flusso di output, come specificato dal parametro intero *b*. La seconda versione di write() utilizza come parametro un array di byte che vengono scritti nel flusso di output. L'ultima versione utilizza come parametri un array di byte, un offset intero e una lunghezza. Questa versione di write() è molto simile alla seconda versione, a eccezione del fatto che utilizza gli altri parametri per determinare in quale punto dell'array di byte iniziare l'output dei dati e quanti dati inviare. Il parametro *off* specifica l'offset nell'array di byte da cui si desidera iniziare l'output dei dati, mentre il parametro *lun* specifica quanti byte devono essere inviati.

Il metodo flush() viene utilizzato per svuotare il flusso di output. Richiamando flush() si obbliga l'oggetto OutputStream a inviare tutti i dati in sospeso.

Infine, il metodo close() chiude un flusso di output e rilascia tutte le risorse associate. Come con gli oggetti InputStream, di norma non è necessario richiamare close() su un oggetto OutputStream, in quanto i flussi vengono chiusi automaticamente quando sono distrutti.

La classe PrintStream

La classe PrintStream deriva da OutputStream ed è stata progettata principalmente per stampare i dati di output come testo. PrintStream ha due costruttori:

- PrintStream(OutputStream *out*)
- PrintStream(OutputStream *out*, boolean *autoflush*)

Entrambi questi costruttori di PrintStream utilizzano come primo parametro un oggetto OutputStream. L'unica differenza tra i due è il modo in cui viene gestito il carattere di nuova riga. Nel primo costruttore, il flusso viene svuotato sulla base di una decisione interna dell'oggetto, mentre nel secondo è possibile specificare tramite il parametro booleano *autoflush* che il flusso venga svuotato ogni volta che si incontra un carattere di nuova riga.

La classe PrintStream implementa inoltre un'ampia serie di metodi, indicati di seguito:

- boolean checkError()
- void print(Object *obj*)
- synchronized void print(String *s*)

- ✔ `synchronized void print(char s[])`
- ✔ `void print(char c)`
- ✔ `void print(int i)`
- ✔ `void print(long l)`
- ✔ `void print(float f)`
- ✔ `void print(double d)`
- ✔ `void print(boolean b)`
- ✔ `void println()`
- ✔ `synchronized void println(Object obj)`
- ✔ `synchronized void println(String s)`
- ✔ `synchronized void println(char s[])`
- ✔ `synchronized void println(char c)`
- ✔ `synchronized void println(int i)`
- ✔ `synchronized void println(long l)`
- ✔ `synchronized void println(float f)`
- ✔ `synchronized void println(double d)`
- ✔ `synchronized void println(boolean b)`

Il metodo `checkError()` svuota il flusso e indica se si è verificato o meno un errore. Il valore restituito da `checkError()` si basa sul fatto che nel flusso sia avvenuto o meno un errore, vale a dire che, dopo che è si è verificato un errore, `checkError()` restituisce sempre `true` per quel flusso.

`PrintStream` contiene numerosi metodi `print()` che gestiscono tutte le esigenze di stampa. La versione di `print()` che utilizza un parametro `Object` stampa semplicemente i risultati della chiamata al metodo `toString()` sull'oggetto. Tutti gli altri metodi `print()` utilizzano un parametro di tipo diverso che specifica quale tipo di dati deve essere stampato.

I metodi `println()` implementati da `PrintStream` sono molto simili ai metodi `print()`; l'unica differenza è che i metodi `println()` stampano i caratteri di nuova riga in base ai dati, mentre il metodo `println()` che non utilizza parametri stampa i caratteri di nuova riga autonomamente.

L'oggetto System.out

Il monitor è il dispositivo principale di output nei computer moderni. La classe `System` ha una variabile membro che rappresenta il flusso di output standard, che di norma è il moni-

tor. La variabile membro è chiamata out ed è un'istanza della classe `PrintStream`. Questa variabile membro, già vista nella maggior parte dei programmi di esempio sviluppati finora, è molto utile per stampare testo sullo schermo.

La classe BufferedOutputStream

La classe `BufferedOutputStream` è molto simile alla classe `OutputStream`, a eccezione del fatto che fornisce un flusso di output *bufferizzato*. Questa classe permette di scrivere in un flusso senza che nel dispositivo di output arrivino troppi dati, mantenendo un buffer che viene riempito quando si scrive nel flusso. Quando il buffer è pieno o viene svuotato esplicitamente, il flusso viene scritto nel dispositivo di output. Questo tipo di approccio è più efficiente, in quanto la maggior parte del trasferimento dei dati avviene nella memoria e l'invio dei dati in un dispositivo viene eseguito un'unica volta.

La classe `BufferedOutputStream` implementa gli stessi metodi definiti in `OutputStream`, cioè non vi sono altri metodi, a eccezione dei due costruttori, indicati di seguito:

- ✔ `BufferedOutputStream(OutputStream` *`out`*`)`
- ✔ `BufferedOutputStream(OutputStream` *`out`*`, int` *`dim`*`)`

Entrambi i costruttori di `BufferedOutputStream` utilizzano come primo parametro un oggetto `OutputStream`. L'unica differenza è data dalle dimensioni del buffer interno utilizzato per memorizzare i dati di output. Nel primo costruttore viene utilizzato un buffer con dimensioni predefinite di 512 byte, mentre nel secondo le dimensioni del buffer vengono specificate con il parametro intero *`size`*. Il buffer stesso all'interno della classe `BufferedOutputStream` è gestito da due variabili membro:

- ✔ `byte` *`buf`*`[]`
- ✔ `int` *`count`*

La variabile membro dell'array di byte *`buf`* è il buffer di dati in cui vengono memorizzati i dati di output, mentre la variabile membro *`conta`* tiene il conto di quanti byte sono memorizzati nel buffer. Queste due variabili membro sono sufficienti per rappresentare lo stato del buffer del flusso di output.

La classe `BufferedWriter` è una versione che si basa su caratteri della classe `BufferedOutputStream`. Il Listato 11.7 include il programma `WriteStuff`, che utilizza un oggetto `BufferedWriter` per stampare un array di caratteri con dati di testo. `WriteStuff` si trova nel file `WriteStuff.java` nel CD-ROM allegato al libro.

Listato 11.7 *La classe WriteStuff.*

```
import java.io.*;
class WriteStuff {
  public static void main (String args[]) {
    // Copia la stringa in un array di byte
    String s = new String("Balla, ragno!\n");
```

```
      char[] buf = new char[64];
      s.getChars(0, s.length(), buf, 0);
      // Esegue l'output dell'array (bufferizzato)
      Writer out = new BufferedWriter(new OutputStreamWriter(System.out));
      try {
        out.write(buf, 0, 64);
        out.flush();
      }
      catch (Exception e) {
        System.out.println("Errore: " + e.toString());
      }
    }
  }
```

Il programma `WriteStuff` riempie un array di caratteri con dati di testo da una stringa e stampa l'array sullo schermo utilizzando un flusso di output bufferizzato. `WriteStuff` inizia con la creazione di un oggetto `String` che contiene del testo e di un array di caratteri. Il metodo `getChars()` di `String` viene utilizzato per copiare i caratteri della stringa nell'array di caratteri. Quando l'array di caratteri è pronto, viene creato un oggetto `BufferedWriter` passando `System.out` nel costruttore di un oggetto `OutputStreamWriter`, che a sua volta viene passato al costruttore dell'oggetto `BufferedWriter`. L'array di caratteri viene quindi scritto nel buffer di output utilizzando il metodo `write()`. Poiché il flusso è bufferizzato, è necessario richiamare il metodo `flush()` per stampare effettivamente i dati.

La classe DataOutputStream

La classe `DataOutputStream` è utile per scrivere i tipi di dati primitivi di Java in un flusso di output in un modo che consente la portabilità. `DataOutputStream` ha un unico costruttore, che utilizza come unico parametro un oggetto `OutputStream`. Questo costruttore è definito nel seguente modo:

`DataOutputStream(OutputStream out)`

La classe `DataOutputStream` implementa i seguenti metodi, oltre a quelli ereditati da `OutputStream`:

- ✔ `final int size()`
- ✔ `final void writeBoolean(boolean b)`
- ✔ `final void writeByte(int i)`
- ✔ `final void writeShort(int i)`
- ✔ `final void writeChar(int i)`
- ✔ `final void writeInt(int i)`
- ✔ `final void writeLong(long l)`
- ✔ `final void writeFloat(float f)`

- `final void writeDouble(double d)`
- `final void writeBytes(String s)`
- `final void writeChars(String s)`

Il metodo `size()` viene utilizzato per determinare quanti byte sono già stati scritti nel flusso; il valore intero restituito da `size()` specifica il numero di byte scritti.

Gli altri metodi implementati in `DataOutputStream` sono tutti varianti del metodo `write()`; ogni versione di `writeType()` utilizza un tipo di dati diverso, che a sua volta viene scritto come output.

La classe FileOutputStream

La classe `FileOutputStream` permette di eseguire operazioni di output su file semplici. Per le operazioni di output su file più avanzate, si utilizza la classe `RandomAccessFile`, discussa più avanti in questo capitolo. È possibile creare un oggetto `FileOutputStream` utilizzando uno dei seguenti costruttori:

- `FileOutputStream(String nome)`
- `FileOutputStream(File file)`
- `FileOutputStream(FileDescriptor fdObj)`

Il primo costruttore utilizza un parametro `String`, che specifica il nome del file da utilizzare per l'output. Il secondo costruttore utilizza come parametro un oggetto `File`, che specifica il file di output (l'oggetto `File` viene descritto più avanti in questo capitolo). Il terzo costruttore utilizza come unico parametro un oggetto `FileDescriptor`.

La classe `FileWriter` funziona esattamente come la classe `FileOutputStream`, a eccezione del fatto che è progettata specificatamente per operare con caratteri, anziché con byte. Il Listato 11.8 include il programma `WriteFile`, che utilizza la classe `FileWriter` per scrivere l'input dell'utente in un file di testo. `WriteFile` si trova nel file `WriteFile.java` nel CD-ROM allegato al libro.

Listato 11.8 *La classe WriteFile.*

```
import java.io.*;
class WriteFile {
  public static void main (String args[]) {
    // Legge l'input dell'utente
    char buf[] = new char[64];
    try {
      Reader in = new InputStreamReader(System.in);
      in.read(buf, 0, 64);
    }
    catch (Exception e) {
      System.out.println("Errore: " + e.toString());
    }
```

```
      // Scrive i dati su un file
      try {
        Writer out = new FileWriter("Output.txt");
        out.write(buf, 0, 64);
        out.flush();
      }
      catch (Exception e) {
        System.out.println("Errore: " + e.toString());
      }
    }
  }
```

In WriteFile, l'input dell'utente viene letto dal flusso di input standard in un array di caratteri utilizzando il metodo read() di InputStreamReader, quindi viene creato un oggetto FileWriter con il nome di file Output.txt, che viene passato come unico parametro del costruttore. Viene quindi utilizzato il metodo write() per stampare l'array di caratteri nel flusso.

Come si può vedere, lavorare con le classi di scrittura è semplice come lavorare con le classi di lettura.

Classi di gestione dei file

Se le classi FileInputStream e FileOutputStream non soddisfano le aspettative di gestione dei file, in Java vi sono due altre classi che operano sui file e che sicuramente soddisfano ogni esigenza: File e RandomAccessFile. La classe File riproduce il modello di una voce visualizzata nell'elenco della directory del sistema operativo, fornendo l'accesso alle informazioni su un file, inclusi gli attributi e il percorso completo in cui si trova. La classe RandomAccessFile, invece, contiene numerosi metodi per leggere e scrivere dati da e in un file.

La classe File

La classe File può essere istanziata utilizzando uno dei seguenti tre costruttori:

- ✔ File(String *percorso*)
- ✔ File(String *percorso*, String *nome*)
- ✔ File(File *dir*, String *nome*)

Il primo costruttore utilizza un unico parametro String che specifica il nome completo del percorso del file. Il secondo costruttore utilizza due parametri stringa: *percorso* e *nome*: il primo specifica il percorso in cui si trova il file, mentre il secondo specifica il nome del file. Il terzo costruttore è simile al secondo, a eccezione del fatto che utilizza un altro oggetto File come primo parametro, anziché una stringa. L'oggetto File in questo caso viene utilizzato per specificare il percorso del file.

I metodi più importanti implementati dalla classe File sono:

- String getName()
- String getPath()
- String getAbsolutePath()
- String getParent()
- boolean exists()
- boolean canWrite()
- boolean canRead()
- boolean isFile()
- boolean isDirectory()
- boolean isAbsolute()
- long lastModified()
- long length()
- boolean mkdir()
- boolean mkdirs()
- boolean renameTo(File *dest*)
- boolean delete()
- String[] list()
- String[] list(FilenameFilter *filtro*)

Il metodo getName() ottiene il nome di un file e lo restituisce come stringa. Il metodo getPath() restituisce il percorso di un file, che potrebbe essere relativo, sotto forma di stringa. Il metodo getAbsolutePath() restituisce il percorso assoluto di un file. Il metodo getParent() restituisce la directory genitore di un file oppure null se non viene trovata una directory genitore.

Il metodo exists() restituisce un valore booleano che specifica se un file esiste effettivamente. I metodi canWrite() e canRead() restituiscono valori booleani che specificano se è possibile scrivere in un file o leggere da un file. I metodi isFile() e isDirectory() restituiscono valori booleani che specificano se un file è valido e se le informazioni sulla directory sono valide. Il metodo isAbsolute() restituisce un valore booleano che specifica se un nome di file è assoluto.

Il metodo lastModified() restituisce un valore long che specifica la data in cui un file è stato modificato l'ultima volta; il valore long restituito è utile solo per determinare il tempo trascorso dalle modifiche, non ha significato come momento assoluto e non è adatto per l'output. Il metodo length() restituisce la lunghezza di un file in byte.

Il metodo `mkdir()` crea una directory sulla base delle informazioni sul percorso corrente e restituisce un valore booleano che indica se la creazione della directory è giunta a buon fine. Il metodo `mkdirs()` è simile a `mkdir()`, a eccezione del fatto che può essere utilizzato per creare un'intera struttura di directory. Il metodo `renameTo()` modifica il nome di un file in base al nome specificato dall'oggetto `File` passato come parametro *`dest`*. Il metodo `delete()` elimina un file. Entrambi i metodi `renameTo()` e `delete()` restituiscono un valore booleano per indicare se l'operazione è giunta a buon fine.

Infine, i metodi `list()` dell'oggetto `File` ottengono un elenco del contenuto della directory. Entrambi i metodi `list()` restituiscono un elenco di nomi di file in un array `String`; l'unica differenza tra i due metodi è che la seconda versione utilizza un oggetto `FilenameFilter` che permette di filtrare determinati file nell'elenco.

Il Listato 11.9 include il codice sorgente del programma `FileInfo`, che utilizza un oggetto `File` per determinare le informazioni su un file nella directory corrente. Il programma `FileInfo` si trova nel file sorgente `FileInfo.java` nel CD-ROM allegato al libro.

Listato 11.9 *La classe FileInfo.*

```
import java.io.*;
class FileInfo {
  public static void main (String args[]) {
    System.out.println("Inserire nome del file: ");
    char c;
    StringBuffer buf = new StringBuffer();
    try {
      Reader in = new InputStreamReader(System.in);
      while ((c = (char)in.read()) != '\n')
        buf.append(c);
    }
    catch (Exception e) {
      System.out.println("Errore: " + e.toString());
    }
    File file = new File(buf.toString());
    if (file.exists()) {
      System.out.println("Nome file  : " + file.getName());
      System.out.println(" Percorso  : " + file.getPath());
      System.out.println("Perc. Ass. : " + file.getAbsolutePath());
      System.out.println("Scrittura  : " + file.canWrite());
      System.out.println("Lettura    : " + file.canRead());
      System.out.println("Lunghezza  : " + (file.length() / 1024) + "KB");
    }
    else
      System.out.println("Spiacente, file non trovato.");
  }
}
```

Il programma `FileInfo` utilizza l'oggetto `File` per ottenere informazioni su un file nella directory corrente. All'utente viene richiesto di digitare un nome di file; l'input viene quindi memorizzato in un oggetto `String`, che viene utilizzato come parametro del costruttore

dell'oggetto File. La chiamata al metodo exists() determina se il file esiste effettivamente, nel qual caso vengono ottenute le informazioni sul file per mezzo di vari metodi File() e i risultati vengono stampati sullo schermo.

Di seguito sono riportati i risultati dell'esecuzione di FileInfo, specificando FileInfo.java quale file per cui si desidera ottenere le informazioni:

```
Nome file  : FileInfo.java
Percorso   : FileInfo.java
Perc. Ass. : C:\Libri\Codice\FileInfo.java
Scrittura  : true
Lettura    : true
Lunghezza  : 0KB
```

La classe RandomAccessFile

La classe RandomAccessFile contiene numerosi metodi per leggere e scrivere in e da file. Nonostante sia certamente possibile utilizzare FileInputStream e FileOutputStream per le operazioni di I/O di file, RandomAccessFile include molte più funzionalità e opzioni. I costruttori di RandomAccessFile sono:

- ✔ RandomAccessFile(String *nome*, String *modo*)
- ✔ RandomAccessFile(File *file*, String *modo*)

Il primo costruttore utilizza un parametro String che specifica il nome del file a cui accedere e un parametro String che specifica il tipo di modalità (lettura o scrittura); il tipo di modalità può essere "r" per la lettura o "rw" per la lettura/scrittura. Il secondo costruttore utilizza un oggetto File come primo parametro, che specifica il file a cui accedere, mentre il secondo parametro è una stringa che indica la modalità, che funziona esattamente come nel primo costruttore.

La classe RandomAccessFile implementa numerosi metodi per l'I/O dei file. Alcuni dei più comuni sono:

- ✔ int skipBytes(int *n*)
- ✔ long getFilePointer()
- ✔ void seek(long *pos*)
- ✔ int read()
- ✔ int read(byte *b*[])
- ✔ int read(byte *b*[], int *off*, int *lun*)
- ✔ final boolean readBoolean()
- ✔ final byte readByte()
- ✔ final int readUnsignedByte()

- ✔ `final short readShort()`
- ✔ `final int readUnsignedShort()`
- ✔ `final char readChar()`
- ✔ `final int readInt()`
- ✔ `final long readLong()`
- ✔ `final float readFloat()`
- ✔ `final double readDouble()`
- ✔ `final String readLine()`
- ✔ `final void readFully(byte b[])`
- ✔ `final void readFully(byte b[], int off, int lun)`
- ✔ `void write(byte b[])`
- ✔ `void write(byte b[], int off, int lun)`
- ✔ `final void writeBoolean(boolean b)`
- ✔ `final void writeByte(int i)`
- ✔ `final void writeShort(int i)`
- ✔ `final void writeChar(int i)`
- ✔ `final void writeInt(int i)`
- ✔ `final void writeLong(long l)`
- ✔ `void writeFloat(float f)`
- ✔ `void writeDouble(double d)`
- ✔ `void writeBytes(String s)`
- ✔ `void writeChars(String s)`
- ✔ `long length()`
- ✔ `void close()`

Osservando questo elenco, si nota che questi metodi sono familiari; la maggior parte di essi infatti è implementata anche in `FileInputStream` o in `FileOutputStream`. Il fatto che `RandomAccessFile` li riunisca in un'unica classe rappresenta già un vantaggio. Questi metodi funzionano esattamente come nelle classi `FileInputStream` e `FileOutputStream`, ma vi sono altri metodi nuovi implementati in `RandomAccessFile`.

Il primo metodo nuovo è `getFilePointer()`, che restituisce la posizione corrente del puntatore al file come valore `long`. Il *puntatore al file* indica la posizione nel file in cui vengono letti o

scritti i dati successivi. Nella modalità di lettura, il puntatore al file è simile alla puntina di un giradischi o al laser di un lettore di CD. `seek()` è l'altro metodo nuovo su cui è necessario soffermarsi; esso imposta il puntatore al file sulla posizione assoluta specificata dal parametro `long` *pos*. Richiamando `seek()` si muove il puntatore al file nella posizione assoluta specificata dal parametro `long` *pos*. La chiamata a questo metodo per spostare il puntatore al file equivale a spostare la puntina di un giradischi con la mano: in entrambi i casi, il punto di lettura dei dati o della musica viene spostato. Una situazione simile si ha quando si scrivono i dati.

Vale la pena notare che i puntatori ai file di Java non sono puntatori nel senso tradizionale del termine, vale a dire riferimenti diretti a una locazione della memoria. Come si sa, Java non supporta i puntatori alla memoria. I puntatori ai file sono più concettuali: indicano un "punto" specifico in un file.

Il Listato 11.10 include il codice sorgente di `FilePrint`, un programma che utilizza la classe `RandomAccessFile` per stampare un file sullo schermo. Il codice sorgente del programma `FilePrint` si trova nel file `FilePrint.java` nel CD-ROM allegato al libro.

Listato 11.10 *La classe FilePrint.*

```
import java.io.*;
class FilePrint {
  public static void main (String args[]) {
    System.out.println("Immettere nome file: ");
    char c;
    StringBuffer buf = new StringBuffer();
    try {
      Reader in = new InputStreamReader(System.in);
      while ((c = (char)in.read()) != '\n')
        buf.append(c);
      System.out.println(buf.toString());
      RandomAccessFile file = new RandomAccessFile(buf.toString(), "rw");
      while (file.getFilePointer() < file.length())
        System.out.println(file.readLine());
    }
    catch (Exception e) {
      System.out.println("Errore: " + e.toString());
    }
  }
}
```

Il programma `FilePrint` inizia in modo molto simile al programma `FileInfo` del Listato 11.9, nel senso che richiede all'utente di digitare un nome di file e memorizza il risultato in una stringa. Di seguito utilizza la stringa per creare un oggetto `RandomAccessFile` nella modalità lettura/scrittura, che viene specificata passando `"rw"` come secondo parametro del costruttore. Successivamente viene utilizzato un ciclo `while` per chiamare ripetutamente il metodo `readLine()`, finché è stato letto l'intero file. La chiamata a `readLine()` viene eseguita all'interno di una chiamata a `println()`, in modo che ogni riga del file venga stampata sullo schermo.

Riepilogo

In questo capitolo sono stati trattati molti argomenti. Il lato positivo è che si è discusso di tutto ciò che è importante sapere sulle operazioni di I/O in Java e sulle classi più importanti nel package di I/O. Non vi è comunque molto altro all'interno di questo package che non sia stato discusso. Il fatto è che le librerie di classi di Java sono particolarmente ampie e alcune classi sono utili solo in circostanze molto particolari. Lo scopo di questo capitolo è quello di evidenziare le classi e i metodi principali all'interno del package di I/O.

Uno degli impieghi dei flussi di input e output è rappresentato dall'invio e la ricezione di dati in una rete. Il prossimo capitolo affronta il package di rete di Java, `java.net`, descrivendo il supporto integrato per le reti incluse nel package e come questo possa essere utilizzato per creare programmi Java in rete.